Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2

Алгоритмы генетический

По дисциплине: ООП

Обучающийся: Арройо Николас

Факультет: Факультет инфокоммуникационных технологий

Группа: К3220

Направление подготовки: 11.03.02 Инфокоммуникационные технологии и системы связи

Образовательная программа: Программирование в инфокоммуникационных системах

Обучающийся _____ Арройо Н.
(дата) (подпись) (Ф.И.О.)

Руководитель _____ Иванов С. Е.
(дата) (подпись) (Ф.И.О.)

ЦЕЛЬ РАБОТЫ

Изучить генетический алгоритм и реализовать алгоритм средствами ООП для решения Диофантова уравнения.

Основные теоретические сведения

Генетический алгоритм — это метод оптимизации, основанный на принципах естественного отбора. Он работает с популяцией решений, представленных в виде генотипов, и использует операторы кроссинговера, мутации, наследования и селекции для генерации новых решений. На каждом этапе алгоритм отбирает лучшие решения с помощью функции приспособленности. Процесс повторяется до достижения заданного критерия остановки. Такой подход эффективно применяется в различных задачах оптимизации.

Задание на лабораторную работу

Необходимо реализовать средствами ООП генетический алгоритм для решения задачи: найти решение Диофантова (только целые решения) уравнения: a+2b+3c+4d=30, где *a, b, c* и *d* - некоторые положительные целые.

Сначала был создан класс Generation, предназначенный для хранения одной особи популяции. В нем объявлены четыре целочисленные переменные *a*, *b*, *c*, *d*, которые участвуют в вычислении целевой функции, а также переменная *ex_res*, обозначающая ожидаемый результат. Кроме того, используется делегат *Func<int, int, int, int, int> f*, который позволяет сохранить формулу в виде переменной и вызывать её как обычную функцию. Для хранения результата вычисления используется приватное поле *real_res* с

типом *int?* (nullable int), что позволяет проверить, было ли значение уже рассчитано, и избежать повторных вычислений — если значение ещё не было получено, оно вычисляется и сохраняется, иначе возвращается сохранённое. Также была создана функция *RealResult*, реализованная в виде свойства, которая управляет этим процессом: при первом вызове она вычисляет результат функции, а при последующих — возвращает уже сохранённое значение. Эта часть кода показана на рисунке 1.1.

```
class Generation
{
    public int a;
    public int b;
    public int c;
    public int d;
    public static Random r = new Random();
    public int ex_res; //ожидаемый результат
    public Func<int, int, int, int, int> f;
    private int? real_res; //получаемый результат
    3 referencias
    public int RealResult
    {
        get
        {
            if (real_res == null)
            {
                real_res = f(a, b, c, d);
            }
            return (int)real_res;
        }
    }
```

Рисунок 1.1 — Класс Generation, определение переменных и свойство RealResult

Для инициализации параметров особи был использован конструктор Generation, принимающий значения переменных *a*, *b*, *c*, *d*, формулу *f* и ожидаемый результат *ex_res*. Это показано на рисунке 1.2.

```
public Generation(Func<int, int, int, int, int> f, int a, int b, int c, int d, int ex_res)
{
    this.f = f;
    this.a = a;
    this.b = b;
    this.c = c;
    this.d = d;
    this.ex_res = ex_res;
}
```

Рисунок 1.2 — Конструктор Класса Generation

Для создания новой особи на основе двух родителей был реализован статический метод *NewGen*. Внутри метода используется конструкция *switch*, которая выбирает один из трёх случаев кроссинговера (генетической комбинации) — в каждом случае часть параметров берётся от первого родителя (*p1*), а другая часть — от второго (*p2*). Это позволяет генетически комбинировать решения и получать разнообразные особи для следующего поколения. Эта часть кода показана на рисунке 1.3.

```
public static Generation NewGen(Generation p1, Generation p2)
{
    int x = r.Next(1, 4);
    switch (x)
    {
        case 1: return new Generation(p1.f, p1.a, p1.b, p2.c, p2.d, p2.ex_res);
        case 2: return new Generation(p1.f, p1.a, p2.b, p2.c, p2.d, p2.ex_res);
        case 3: return new Generation(p1.f, p1.a, p1.b, p1.c, p2.d, p2.ex_res);
        default: return null;
    }
}
```

Рисунок 1.3 — Метол NewGen

Для реализации мутации особей был создан метод *Change*. Этот метод случайным образом выбирает одну из четырёх переменных (*a*, *b*, *c*, *d*) и заменяет её на новое случайное значение от 1 до 49 включительно. Такая мутация позволяет внести разнообразие в популяцию и избежать зацикливания алгоритма на локальных оптимумах. Эта часть кода показана на рисунке 1.4.

```
public void Change()
{
    int imposter = r.Next(1, 5);
    switch (imposter)
    {
        case 1:
            a = r.Next(1, 50);
            break;
        case 2:
            b = r.Next(1, 50);
            break;
        case 3:
            c = r.Next(1, 50);
            break;
        case 4:
            d = r.Next(1, 50);
            break;
        default:
            break;
    }
}
```

Рисунок 1.4 — Полиномиальный хэш

Для оценки качества каждой особи был реализован метод *GetAc*, который возвращает модуль разности между ожидаемым результатом *ex_res* и фактическим результатом, вычисленным с помощью функции *f*. Чем меньше это значение, тем ближе решение к оптимальному.

Кроме того, был реализован статический метод *SrKoef*, который принимает список особей *List<Generation> generations* — это коллекция текущих решений в популяции — и вычисляет среднее значение оценки приспособленности (среднюю ошибку) по всем особям. Таким образом определяется, насколько хороша текущая популяция, и можно принять решения об эволюции. Методы GetAc и SrKoef показаны на рисунке 1.5.

```
public int GetAc()
{
    return Math.Abs((int)(ex_res - RealResult));
}
2 referencias
public static double SrKoef(List<Generation> generations)
{
    double sum = 0;
    foreach (Generation item in generations)
    {
        sum += item.GetAc();
    }
    return sum / generations.Count();
}
```

Рисунок 1.5 — Методы GetAc и SrKoef

Потом, вклассе Program были объявлены три переменные. Переменная *res* задаёт целевое значение, которое должно быть достигнуто функцией — в данном случае это 30. Далее была объявлена переменная *f*, представляющая собой лямбда-функцию. Наконец, была создана переменная *r* — экземпляр класса Random. Эта часть кода показана на рисунке 1.6.

```
class Program
{
    public static int res = 30;
    public static Func<int, int, int, int, int> f = (a, b, c, d) => a + 2 * b + 3 * c + 4 * d;
    public static Random r = new Random();
```

Рисунок 1.6 — Объявление переменных класса Program

В методе Main сначала создаётся список *generations*, содержащий начальную популяцию из пяти случайно сгенерированных особей. Каждая особь создаётся с помощью конструктора *Generation*, в который передаётся функция *f*, а также случайные значения параметров *a*, *b*, *c*, *d* от 1 до 29. После этого создаётся копия этой популяции в переменной *initialPopulation*, которая будет использоваться позже для вывода и сравнения результатов. Эта часть кода показана на рисунке 1.7.

```
static void Main(string[] args)
{
    List<Generation> generations = new List<Generation>();
    for (int i = 0; i < 5; i++)
    {
        generations.Add(new Generation(f, r.Next(1, 30), r.Next(1, 30), r.Next(1, 30), r.Next(1, 30), res));
    }

    List<Generation> initialPopulation = generations
        .Select(g => new Generation(g.f, g.a, g.b, g.c, g.d, g.ex_res))
        .ToList();
```

Рисунок 1.7 — Генерация начальной популяции и создание копии

После инициализации популяции начинается основной цикл работы алгоритма, реализованный с помощью бесконечного цикла *for*. На каждой итерации происходит вывод текущей популяции и расчёт средней ошибки с использованием метода *SrKoef*. Затем особи сортируются по значению функции приспособленности, и выбираются лучшие решения. Если среди них находится хотя бы одно решение, которое даёт точный результат (*f(...) == res*), вызывается метод Solution, и алгоритм завершает свою работу, как показано на рисунке 1.8.

```
for (int i = 0; ; i++)
{
    Console.WriteLine("Iteration " + (i + 1));
    foreach (var item in generations)
    {
        Console.WriteLine("a = " + item.a + " | b = " + item.b + " | c = " + item.c + " | d = " + item.d);
    }
    Console.WriteLine("---");

    double koef1 = Generation.SrKoef(generations);
    var best = generations
        .OrderBy(g => g.GetAc())
        .ToList();
    var match = best.FirstOrDefault(g => g.RealResult == res);
    if (match != null)
    {
        Solution(match, i);
        break;
    }
```

Рисунок 1.8 — Проверка условий завершения и выбор лучших особей

Если нужного решения пока не найдено, создаётся новое поколение из пяти особей, комбинируя гены трёх лучших предыдущих решений с помощью метода *NewGen*. Эти особи добавляются в список *generations*, предварительно

очищенный. Далее рассчитывается средняя ошибка новой популяции (*koef2*) и сравнивается со старой (*koef1*). Если новая популяция не показала улучшения, к случайному количеству особей применяется мутация через метод *Change*, чтобы повысить разнообразие решений и избежать застоя. Эта часть кода показана на рисунке 1.9.

```
        generations.Clear();
        generations.Add(Generation.NewGen(best[0], best[1]));
        generations.Add(Generation.NewGen(best[1], best[0]));
        generations.Add(Generation.NewGen(best[0], best[2]));
        generations.Add(Generation.NewGen(best[2], best[0]));
        generations.Add(Generation.NewGen(best[1], best[2]));

        double koef2 = Generation.SrKoef(generations);
        if (koef2 <= koef1)
        {
            int x = r.Next(1, 5);
            for (int j = 0; j < x; j++)
            {
                generations[r.Next(0, 5)].Change();
            }
        }
    }

    PrintPopulation(initialPopulation, "Initial Population");
    PrintPopulation(generations, "Final Population");
```

Рисунок 1.9 — Создание нового поколения и применение мутации

Когда находится особь, дающая точное значение целевой функции (*RealResult == res*), вызывается метод *Solution*, который выводит на экран значения параметров. Дополнительно указывается, на какой итерации была найдена точная особь. Эта часть кода показана на рисунке 1.10.

```
public static void Solution(Generation g, int iteration)
{
    Console.WriteLine("\nSolution found in " + (iteration + 1) + " iterations:");
    Console.WriteLine("a = " + g.a + " | b = " + g.b + " | c = " + g.c + " | d = " + g.d);

    int part1 = g.a;
    int part2 = 2 * g.b;
    int part3 = 3 * g.c;
    int part4 = 4 * g.d;
    int result = part1 + part2 + part3 + part4;

    Console.WriteLine("\nValue verification:");
    Console.WriteLine($"{g.a} + 2 * {g.b} + 3 * {g.c} + 4 * {g.d} |=> {part1} + {part2} + {part3} + {part4} = {result}\n");
}
```

Рисунок 1.10 — Метод Solution

В завершение работы алгоритма вызывается метод *PrintPopulation*, который выводит параметры и значения функции для каждой особи в популяции. Метод вызывается дважды: сначала для отображения начальной популяции (*initialPopulation*), затем для финальной популяции (*generations*) после выполнения всех итераций. Это позволяет визуально сравнить, какие изменения произошли в популяции в процессе эволюции до нахождения точного решения. Эта часть кода показана на рисунке 1.11.

```
public static void PrintPopulation(List<Generation> gens, string title)
{
    Console.WriteLine($"\n{title}");
    foreach (var g in gens)
    {
        Console.WriteLine($"a = {g.a} | b = {g.b} | c = {g.c} | d = {g.d} | Value = {g.RealResult}");
    }
}
```

Рисунок 1.11 — Метод PrintPopulation

На рисунке 1.12 показан пример работы программы.

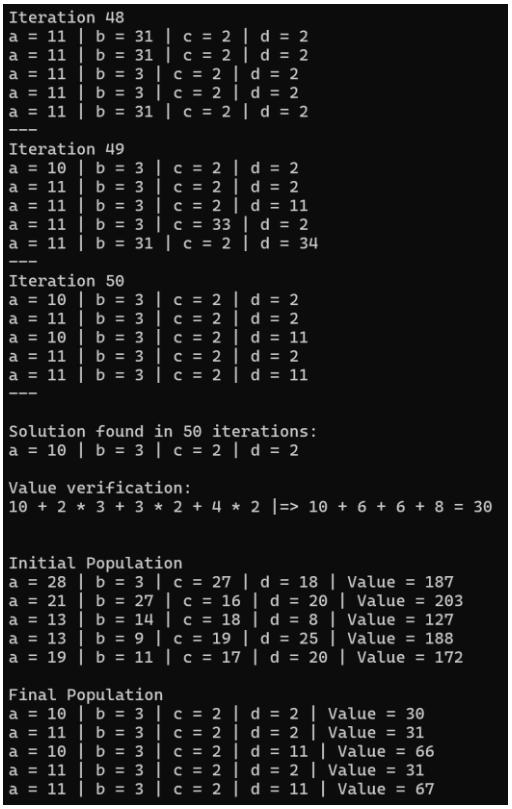

Рисунок 1.12 — Метод PrintPopulation

В блоке *Initial Population* можно увидеть, что начальные особи имеют большие значения функции (Value), значительно превышающие целевое значение 30, например 203, 188 или 172. Это говорит о том, что изначально решения далеки от оптимума.

После 50 итераций алгоритм находит точное решение: a = 10, b = 3, c = 2, d = 2, дающее значение функции 30. В блоке *Final Population* видно, что значения функции у всех особей существенно ближе к целевому, и одна из них полностью удовлетворяет условию (Value = 30). Остальные особи в финальной популяции — это мутации и вариации вокруг найденного решения, с небольшим отклонением от цели (Value = 31, 66, 67). Это подтверждает эффективность алгоритма в постепенном приближении к оптимальному решению путём эволюции.

Контрольные вопросы

1. Какие конструкции ООП C# применяются для реализации генетического алгоритма?

Использовались классы (Generation, Program), инкапсуляция (приватные поля и свойства), делегаты (Func<> для хранения функции), а также списки (List<Generation>). Это обеспечивает модульность, переиспользуемость и читаемость кода.

2. Как оценивается сложность генетического алгоритма?

Сложность зависит от размера популяции и числа итераций. В худшем случае — O(n * g), где n — количество особей, g — количество поколений. На практике алгоритм может быть эффективнее благодаря быстрой сходимости.

3. Какие задачи решаются с помощью генетического алгоритма?

Задачи оптимизации, подбора параметров, планирования, маршрутизации, машинного обучения и другие, где невозможно найти точное решение аналитически или перебором.

4. Какие ограничения и условия на применение алгоритма?

Необходима функция приспособленности, которую можно эффективно вычислить. Алгоритм не гарантирует глобального оптимума, чувствителен к параметрам (размер популяции, вероятность мутации) и может застрять в локальном оптимуме.

5. Какие определяются операторы кроссинговера, наследования, мутаций и селекции?

В данной реализации:

- Кроссинговер — комбинирование генов двух родителей;
- Наследование — передача значений от родителей к потомку;
- Мутация — случайное изменение одного гена;
- Селекция — отбор лучших решений по функции приспособленности.

6. Как можно оценить точность генетического алгоритма?

По значению функции приспособленности: чем ближе результат к целевому, тем точнее решение. Также можно анализировать среднюю ошибку популяции и количество итераций до нахождения решения. Генетический алгоритм — это метод оптимизации, основанный на принципах естественного отбора. Он работает с популяцией решений, представленных в виде генотипов, и использует операторы кроссинговера, мутации, наследования и селекции для генерации новых решений. На каждом этапе алгоритм отбирает лучшие решения с помощью функции приспособленности. Процесс повторяется до достижения заданного критерия остановки. Такой подход эффективно применяется в различных задачах оптимизации.

ЗАКЛЮЧЕНИЕ

В процессе выполнения работы я изучил принципы работы генетического алгоритма и реализовал его средствами объектно-ориентированного программирования на языке C# для решения диофантова уравнения a + 2b + 3c + 4d = 30. Я подробно ознакомился с ключевыми компонентами алгоритма: генерацией начальной популяции, функцией приспособленности, отбором, операциями кроссинговера и мутации.

На практике был создан класс Generation, инкапсулирующий структуру особи и реализующий методы для вычисления результата, оценки точности, создания новых особей и мутации. С помощью бесконечного цикла в методе Main особи эволюционировали от случайных значений к точному решению. Алгоритм завершался только при нахождении особи с результатом, точно равным 30, что гарантирует его корректность.

Работа над проектом позволила углубить понимание ООП в C#, в том числе работу с классами, свойствами, делегатами, списками и лямбда-выражениями. Также я убедился в эффективности эволюционных методов при решении задач оптимизации с ограничениями.